Concurrent Data Structures
Correctness and Performance
Semantics of concurrent data structures

- **Sequential specification** = set of legal sequences
  - e.g. pools, queues, stacks
  - e.g. queue legal sequence: enq(1)enq(2)deq(1)deq(2)

- **Consistency condition** = e.g. linearizability / sequential consistency
  - e.g. the concurrent history above is a linearizable queue concurrent history
Consistency conditions

Linearizability  [Herlihy, Wing ’90]

there exists a legal sequence that preserves precedence

t1:

\text{enq(2)}^2 \quad \text{deq(1)}^3

t2:

1\text{enq(1)} \quad \text{deq(2)}^4

Sequential Consistency  [Lamport’79]

there exists a legal sequence that preserves per-thread precedence (program order)

t1:

1\text{enq(1)} \quad \text{deq(2)}^4

t2:

\text{deq(1)}^2 \quad \text{enq(2)}^3
Performance and scalability
Relaxations allow trading correctness for performance.

Provide the potential for better-performing implementations.
Relaxing the Semantics

- **Sequential specification** = set of legal sequences
- **Consistency condition** = e.g. linearizability / sequential consistency

Quantitative relaxations - POPL13
Henzinger, Kirsch, Payer, Sezgin, S.

Local linearizability - CONCUR16
in this talk

not “sequentially correct”

for queues only (feel free to ask for more)

too weak
Local Linearizability

main idea

- **Partition** a history into a set of local histories
- **Require** linearizability per local history

Already present in some shared-memory consistency conditions (not in our form of choice)

Local sequential consistency… is also possible

no global witness
Local Linearizability (queue) example

- **t1:**
  - enq(1)
  - deq(2)

- **t2:**
  - enq(2)
  - deq(1)

- (sequential) history not linearizable

- t2-induced history, linearizable

- t1-induced history, linearizable

- locally linearizable
Local Linearizability (queue) definition

Queue signature $\Sigma = \{\text{enq}(x) \mid x \in V\} \cup \{\text{deq}(x) \mid x \in V\} \cup \{\text{deq}(\text{empty})\}$

For a history $h$ with a thread $T$, we put

$\text{I}_T = \{\text{enq}(x)^T \in h \mid x \in V\}$

$\text{O}_T = \{\text{deq}(x)^{T'} \in h \mid \text{enq}(x)^T \in \text{I}_T\} \cup \{\text{deq}(\text{empty})\}$

$h$ is locally linearizable iff every thread-induced history $h_T = h \mid (\text{I}_T \cup \text{O}_T)$ is linearizable.
Where do we stand?

In general

Local Linearizability

Linearizability

Sequential Consistency
Where do we stand?

For queues (and most container-type data structures)

Local Linearizability

Linearizability

Sequential Consistency
Local linearizability is compositional

\( h \) (over multiple objects) is locally linearizable iff each per-object subhistory of \( h \) is locally linearizable

Local linearizability is modular / “decompositional”

uses decomposition into smaller histories, by definition

may allow for modular verification

like linearizability

unlike sequential consistency
Generic Implementations

Your favorite linearizable data structure implementation

turns into a locally linearizable implementation by:

- $\Phi$
- Segment of possibly dynamic size ($n$)
- Local inserts / global (randomly distributed) removes

- $LLD \Phi$ (locally linearizable)
- $LL+D \Phi$ (also pool linearizable)
Performance

(a) Queues, LL queues, and “queue-like” pools

LL+D MS queue performs significantly better than MS queue
Local linearizability utilizes the idea of decomposing a history into a set of thread-induced histories and requiring consistency of all such, yielding an intuitive and verifiable consistency condition. For ease of presentation, we define two more sets. First, given a thread index $i$, and a set of insert operations $\mathcal{I}$ we define $\mathcal{H}_i(\mathcal{I})$ to be the set that contains all insert-operations that appear in $\mathcal{I}$ but are performed in thread $i$. Second, given a set $\mathcal{H}$ of histories, we define $\mathcal{H}_i(\mathcal{H})$ to be the set that contains for each insert operation in $\mathcal{H}$, an observation that is performed in thread $i$. We separate in $\mathcal{H}_i(\mathcal{H})$ the set of operations that appear in $\mathcal{H}$ by adding $\mathcal{H}_i(\mathcal{H}) = \{ p \in \mathcal{H} | \text{operation } p \text{ is performed in thread } i \}$ for each thread $i$.

There are different answers to this question. (1) Program-aware performance; i.e., identifying classes of programs that are (in)sensitive to notions of (relaxed) semantics. (2) Program-aware correctness, i.e., identifying classes of programs that are (in)sensitive to the correctness of a given implementation into a locally linearizable one, resulting in improvements of performance and scalability. There are at least two generic implementation schemes that turn a linearizable data structure synchronization at the program level. For example, queues which can relax their semantics if operations are performed out-of-order queues only have one fixed order of operations and a history $\mathcal{H}$ of observations in history $\mathcal{H}$. By adding $\mathcal{H}_i(\mathcal{H})$ to the definition of $\mathcal{H}$, we can impose synchronization on all observers, i.e., add the observation $a$ to the history $\mathcal{H}$ when it performs $a$. However, if the programmer does not explicitly define $\mathcal{H}$, then the insertion of $a$ will be in thread $i$.

The set $\mathcal{H}$ contains for each insert operation in $\mathcal{H}$ a corre-

(2 hyperthreads per core) machine crobenchmarks with an increasing number of threads on a 40-core processor for the performance and scalability of producer-consumer microbenchmarks. Figure 8: Performance and scalability of producer-consumer microbenchmarks. (a) Queues, LL queues, and “queue-like” pools. (b) Stacks, LL stacks, and “stack-like” pools.
Performance

![Graph showing performance and scalability of producer-consumer microbenchmarks with an increasing number of threads on a 40-core machine.](image)

**Figure 8:** Performance and scalability of producer-consumer microbenchmarks with an increasing number of threads on a 40-core machine.

- **LL+D MS** queue performs better than the best known pools.

(a) Queues, LL queues, and “queue-like” pools.
Local Linearizability

Ana Sokolova

joint work with:

Andreas Haas  Google
Andreas Holzer
Michael Lippautz  Google
Ali Sezgin

Tom Henzinger
Christoph Kirsch
Hannes Payer  Google
Helmut Veith